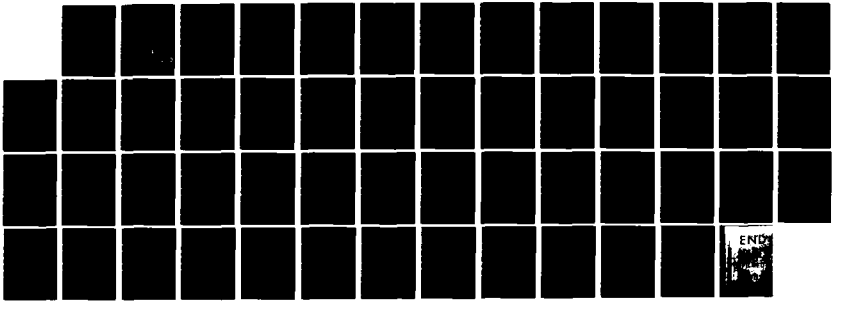
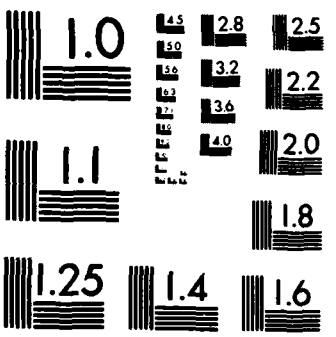
AD-A134 070 COMPUTER PROGRAM DEVELOPMENT SPECIFICATION FOR ADA INTEGRATED ENVIRONMENT. (U) INTERMETRICS INC CAMBRIDGE MA 05 JAN 83 IR-682-1 F30602-80-C-0291 1/1

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(1)

IR-682-1
COMPUTER PROGRAM DEVELOPMENT SPECIFICATION FOR Ada INTEGRATED ENVIRONMENT: MAPSE DEBUGGING FACILITIES B5-AIE (1). DBUG (1)
5 JANUARY 1983

CONTRACT F30602-80-C-0291

PREPARED FOR: ROME AIR DEVELOPMENT CENTER
CONTRACTING DIVISION/PKRD
GRIFFISS AFB, N.Y. 13441

PREPARED BY: INTERMETRICS, INC.
733 CONCORD AVE.
CAMBRIDGE, MA 02138

This document was produced under contract F30602-80-C-0291/ SAP0009 for the Rome Air Development Center. Mr. Donald Mark is the Program Engineer for the Air Force. Mr. Mike Ryer is the Project Manager for Intermetrics.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont'd.)

TABLE OF CONTENTS (Cont'd.)

LEFT BLANK INTENTIONALLY

1.0 SCOPE

This document specifies the requirements for the performance and verification of the AIE debugging facilities. It includes descriptions of: (1) the user debug command language; (2) the DEBUGGER (DBUG) subsystem that provides these facilities; and (3) the interface between DBUG and other AIE components, through which debugging tasks are performed.

1.1 Identification

DBUG is classified within the AIE configuration both as a subsystem and as a CPCI. It consists of the following CPC's.

COMMAND PROCESSOR;	(A)
EXECUTION CONTROL PROCEDURES;	(B)
BREAKPOINT COMMAND PROCEDURES;	(C)
UTILITY PROCEDURES;	(D)
INFORMATION COMMAND PROCEDURES;	(E)
PROGRAM LIBRARY ACCESS PROCEDURES;	(F)
DBUG DATABASE.	(G)

1.2 Functional Summary

DBUG provides facilities for dynamic, symbolic debugging of user programs within the AIE. A program is run under DBUG control with user-specified points at which execution is suspended (breakpoints). At such points, the user can examine and modify program variables, trace and modify the flow of execution, and establish additional breakpoints. The user specifies debugging actions via a command language that permits ordinary Ada representations for most program elements. Commands can be issued interactively or stored as scripts for batch mode processing.

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

LEFT BLANK INTENTIONALLY

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

2.0 APPLICABLE DOCUMENTS

2.1 Program Definition Documents

Reference Manual for the Ada Programming Language, Draft proposed ANSI standard document, July 1982.

Requirements for Ada Programming Support Environments, "STONEMAN", February 1980, Department of Defense.

Revised Statement of Work, (15 March 1980).

2.2 Inter Subsystem Specifications

System Specification for Ada Integrated Environment, Type A, AIE(1).

Computer Program Development Specifications for Ada Integrated Environment (Type B5):

Ada Compiler Phases, AIE(1).COMP(1)

KAPSE/Database, AIE(1).KAPSE(1)

MAPSE Command Processor, AIE(1).MCP(1)

MAPSE Generation and Support, AIE(1).MGS(1)

Virtual Memory Methodology, AIE(1).VMM(2)

Program Integration Facilities, AIE(1).PIF(1)

MAPSE Text Editor, AIE(1).TXED(1)

Technical Report (Interim), IR-684

2.3 Military Specifications and Standards

Data item description DI-E-30139, USAF, 24 July 1976.

LEFT BLANK INTENTIONALLY

3.0 REQUIREMENTS

3.1 Introduction

This section provides the set of requirements for DBUG, a symbolic debugging facility within the Ada Integrated Environment (AIE).

3.1.1 General Description

DBUG is a MAPSE tool that allows a user to control an executing program, suspending execution when desired to examine its state. To do this, DBUG requires support from a variety of AIE components. Suspension of program execution within the DBUG environment occurs at preset breakpoints. Modifications to program variables can be made at such breakpoints and execution resumed with these new values. The user specifies debugging actions via a DBUG command language, either interactively or in a script for batch mode processing.

3.1.2 Peripheral Equipment Identification

Not applicable.

3.1.3 Interface Identification

As do all MAPSE tools, DBUG interfaces with the KAPSE [see AIE(1).KAPSE(1)] for all machine-dependent operations. It additionally requires a DBUG-specific interface with the KAPSE Runtime System (KAPSE.RTS) to process breakpoints.

Other significant interfaces are with the Ada compiler [see AIE(1).COMP(1)], which must provide an object module in a format required for debugging purposes and with the Program Integration Facilities [see AIE(1).PIF(1)] whose program library tools support symbolic debugging. These interfaces are described in more detail in 3.2.4.

3.2 Functional Description

3.2.1 Equipment Description

Not applicable.

3.2.2 Computer Input/Output Utilization

Not applicable.

3.2.3 Computer Interface Block Diagram

Not applicable.

3.2.4 Program Interfaces

DBUG is invoked via the MAPSE Command Processor [AIE(1).MCP(1)] with four parameters. The first parameter to DBUG specifies which program DBUG is to control. This argument is a "window" on the context object of the user program. See [AIE(1).KAPSE(1)] for a description of window and context object. This parameter can be either the name of a program that is to begin execution under DBUG or the name of the program context object if the program to be debugged has already begun execution and has been suspended (e.g., if an unhandled exception has been raised or the program has been interrupted). The second parameter is an ASCII string containing the parameters to be passed to the user program being debugged. If the user program has already begun execution when DBUG is called, this second parameter is ignored.

The third and fourth parameters, respectively, specify the source from which DBUG commands are input, and the source to which output is to be directed. The defaults for these are the standard input and standard output devices, namely the user's terminal. In the case that DBUG is being run in the background, these are a script of DBUG commands and a file to write output to; the MCP interface handles I/O direction.

3.2.4.1 Compiler Interface

The Ada compiler has two user-specified parameters that affect the functionality seen by the DBUG user. These are DEBUG and OPTIMIZE and are described in AIE(1).COMP(1).

The DEBUG parameter value of BREAK causes the compiler to insert hooks before each statement in the object code of each module compiled with this directive. There are four types of hooks. Entry and exit hooks mark the entries and exits for subprograms; flow of control hooks mark statements in the program where the flow of control changes; and statement hooks mark the beginnings of all other statements. See Section 3.2.4.2 for a discussion of the processing of hooks. For modules compiled without hooks, the AFTER, STEP and ON_STEP commands, and the FLOW option for the TRACE command are not available.

The ALTER value of the DEBUG parameter restricts the compiler to optimization within statement boundaries. If this option is not specified, and the OPTIMIZE parameter value TIME (or SPACE) is used, the compiler is permitted to move code and do other optimizations. In this case the DBUG Execution Control Commands and the modification of program variables are not guaranteed to have their intended effect. In addition, the GOTO functions are not available.

When a user issues a DBUG command that might not have the intended effect due to the OPTIMIZE or DEBUG parameters of the specific compilation unit, DBUG prints a brief warning message and then will try to process the command. An example of this would be a reference a variable or statement which has been optimized away. In order to use DBUG in complete confidence, the DEBUG (ALTER, BREAK) configuration should be used during compilation.

DBUG checks the options and pragmas used in each compilation unit separately. The most optimized level found is used to determine the restrictions which apply to the unit.

The back end of the compiler (COMP.BE) provides a statement table which is used by the Breakpoint Command Package to set breakpoints. This table has two kinds of records, one for code with hooks and one for code without. The record for hooked code will contain the statement number and address of the hook. The record for unhooked code will contain the statement number, the address of the first instruction for the statement, and the sum of the lengths of those instructions that would have to be replaced by a hook.

3.2.4.2 KAPSE Interface

DBUG control over user program execution is accomplished by communication primitives between the UTILITY PROCEDURES CPC and the DBUG Support Routine (DSR), which is linked to the user program.

Various pieces of information need to be passed between the DSR and DBUG. The following sections describe that information, as well as defining the hook mechanism. Details of the implementation of hooks, and the information passed between DBUG and DSR are described informally.

3.2.4.2.1 DSR Interface

In general, information is passed to and from the DSR as a set of codes representing specific actions to take or indicating events that have occurred, and an object representing a value on the target machine (e.g., the result of a memory fetch requested by DBUG, or the identity of an exception).

(a) Execution Control. DBUG controls execution of the user program by using communication primitives provided by the KAPSE to invoke DSR functions such as continuing program execution and activating breakpoints.

The capability of halting the program at user defined locations, as well as certain of the history trace functions, are facilitated by the presence of hooks. Hooks are special branches to the DSR. These are installed in the object module by the Ada Compiler under control of the DEBUG parameter. When the value of the parameter is BREAK, hooks are installed between every statement of the compilation units affected by the parameter. If the DEBUG (BREAK) parameter was not specified, the user may insert breakpoints at specified statements, but will not be able to invoke some commands that are available with hooks, such as single stepping and maintaining a history trace of flow-of-control constructs.

Both compiler-generated hooks and breakpoints inserted by DBUG are special branches to the DSR. Special care is taken in the DSR and in the compiler code-generating phases to assure that the hooks are as small and as efficient as possible so that an unactivated hook or breakpoint has minimal effect on the executing program. When a program is run under DBUG, the execution of an activated hook or breakpoint will result in the DSR giving control to DBUG to perform the processing associated with the breakpoint. In certain circumstances, such as when single stepping is performed or when some history trace functions have been invoked, unactivated hooks may also pass control to DBUG.

In addition to halting the program, DBUG has the capability of resuming the program at the point at which it was halted or at a user specified statement (e.g., GOTO). To accomplish this, the DSR passes the RTS the address at which execution should resume.

(b) Value Referencing. DBUG can modify and print the values of program variables. DBUG accomplishes this by calling the DSR, supplying an address and a value to be stored there.

(c) Exception Handling. The DSR can cause the RTS to trap specific exceptions, all exceptions, or only unhandled exceptions. The RTS raise handler does not unwind the stack until it finds a handler for the specific exception. (The run-time stack contains information from which the raise handler can compute which stack frames handle which exceptions.) This means that the user's program context is saved when an unhandled exception occurs rather than having the program unwind to top level. Control returns to DBUG and the user can then take corrective action. This exception handling mechanism is always used, even when DBUG is not present at the time of the exception. This permits the user, once his program has terminated due to an unhandled exception, to invoke DBUG, examine his program state, and take corrective action as necessary.

If the user has specified that the program be halted whenever certain exceptions occur, the DSR will inform the RTS to return to the DSR when any exception is raised, indicating what exception has occurred. The DSR will then determine if the particular exception has been specified by the user and, if so, will return control to

DBUG indicating the exception and where it took place. If the exception has not been specified, the DSR will inform the RTS to process the exception in the normal manner.

(d) Tasking Support. DBUG allows the user to control the execution of tasks within a program (see DELAY, PRIORITY and ABORT), as well as to display the status of some or all of them (see DISPLAY TASKS). This function is provided by communication between the DSR and the tasking support package of KAPSE.RTS.

When a program is executing under DBUG, every creation and activation of a task will result in the tasking support package notifying the DSR of the current program location and the task control block associated with the task. The DSR will record this data to be used later in identifying the task while processing breakpoints and exceptions and displaying information to the user. If the user has specified that a trace of tasking events be maintained, the tasking support package will notify the DSR of every tasking event, indicating which event occurred, as well as the current program location.

The tasking support package will also supply, on request from the DSR, access to the various data structures associated with a task including its dependency list, ENTRY and ACCEPT body unit data area, etc., as well as access to the priority queues.

In response to the user setting the %HI_PRIORITY DBUG variable, the tasking support package will be instructed to suspend any task whose priority is less than or equal to the priority value specified when control returns to DBUG.

3.2.4.2.2 Control Interface

The KAPSE provides a mechanism so that when a special character (interrupt) is entered at the user terminal, the current program associated with that terminal is suspended. If the program is being debugged, the KAPSE will return control to DBUG. Otherwise, the KAPSE will return control to the process that initiated the program (e.g., MCP).

3.2.4.3 Program Library Interface

DBUG requires most of the information stored in the Program Library for the program being debugged. This information is accessed through the Program Library Interface Package of the Program Integration Facilities [AIE(1).PIF(1)].

The DIANA for the compilation units of the user's program is accessed directly from the Program Library. It provides symbol table information for the Analyze procedure, storage information for Evaluate, and other information about user program entities such as statements.

The remainder of the Program Library is referenced through CPCs of the Interface Package.

The statement tables are accessed through the Object Identification Package and Object Module Format CPCs. They provide statement address and other information necessary for setting breakpoints and displaying traces. The user program's relocation map, created by the Linker, is accessed through the Library Management Package. It is used to calculate absolute addresses of variables and statements.

The Cross Reference Package supplies cross references for the LIST and MODIFY commands. The Source Reconstruction Package generates lines of source for the LIST command.

3.2.5 Functional Description

DBUG is composed of various Ada subprograms, tasks and packages linked together into an executable program. Figure 3-1 indicates the flow and data interfaces between the components of DBUG that are described below.

COMMAND PROCESSOR accepts user commands in the DBUG Command Language (DCL) and calls other CPCs as necessary for processing. It generates and later interprets the parse trees representing breakpoint scripts. It evaluates DCL expressions, which can include DBUG and user-program variables.

BREAKPOINT COMMAND PROCEDURES contains the subprograms that provide functions needed to create and maintain user specified breakpoints. These functions include reading and writing to a breakpoint table, and placing and removing breakpoint hooks as necessary.

EXECUTION CONTROL PROCEDURES contains the subprograms that provide the functions needed to control the execution of the user program. These functions include stopping at a breakpoint and executing any script that is associated with that breakpoint, controlling the STEP function, providing the GOTO function, and proceeding at the correct user program address.

UTILITY PROCEDURES contains the subprograms that provide functions needed by several of the other DBUG CPCs. These functions include communicating with the program context being debugged, modification of storage and communicating with other programs such as the program integration tools.

INFORMATION COMMAND PROCEDURES provides the functions needed to display information to the user. This information includes the values of program variables displayed according to their type, the announcement of breakpoints when they are encountered and the listing of the program source.

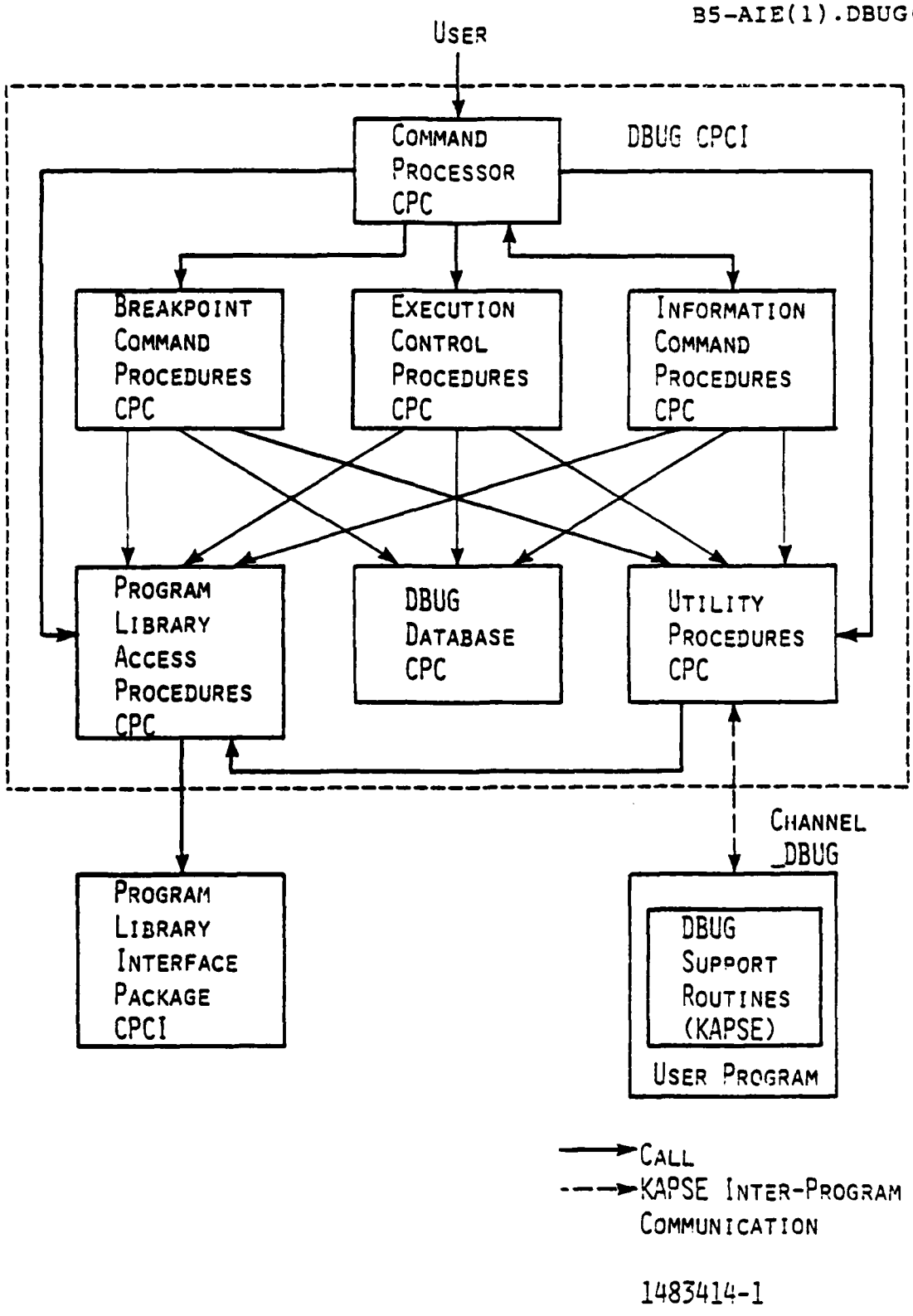


FIGURE 3-1: DBUG Procedures and Interfaces

PROGRAM LIBRARY ACCESS PROCEDURES interfaces with the user-program's program library. They provide access to the symbol and statement tables, linker information, cross reference information and source listings.

DBUG DATABASE controls global DBUG information which must be accessed by more than one CPC. This information includes the breakpoint tables, the breakpoint scripts and the current program state.

3.3 Detailed Functional Requirements

3.3.1 COMMAND PROCESSOR

3.3.1.1 Inputs

Input to COMMAND PROCESSOR is a sequence of one or more commands written in the DBUG Command Language described below. There are four basic categories of commands:

(1) breakpoint commands - these allow the user to set, remove, suspend and restore various breakpoints and actions associated with them.

(2) execution control - these allow the user to stop, start, and modify the execution of his program.

(3) information commands - these allow the user to inspect the state of his program by examining the values of variables, the current location of execution and the history of execution (call chain, task scheduling, rendezvous, jump flow).

(4) DBUG control - these initiate and suspend DBUG processing, read in command files, and redirect output.

3.3.1.1.1 Language Overview

The Debug Command Language (DCL) is based upon a subset of the MCP command language (MCL). The subset excludes multiple streams of execution and database object manipulation. The subset retains the MCL syntax for commands, expressions, script specification and script invocation.

DCL extends MCL in two ways. DBUG-specific commands and predefined variables have been added. In addition, expressions have been generalized to accept user-program variable names in the same contexts as DBUG Command Processor (DCP) variable names. As in MCL, DCP variable names begin with a "%".

A DBUG command generally consists of a keyword specifying an action to be performed and any parameters required by the specific command. When user-program variables are required in a DBUG command, they may be expressed in normal Ada syntax. This includes access dereferencing, array subscripting and record component selection. Ambiguous names may be qualified using normal Ada syntax as well. The expressions used as array subscripts may not include function calls. In addition, operators are not resolved to any overloaded definition. All lists of identifiers provided as parameters to a DBUG command are separated by commas. These include variable names, statement identifiers and exception names. A DBUG command is terminated by semicolon or newline. The stored command part of breakpoint commands which begin with the keyword "BEGIN", is terminated only by the matching "END".

The term 'statement', as used in this document, refers to entities which are assigned a statement number by the Unit Lister [see AIE(1).PIF(1)]. These include simple or compound statements (as described in the Ada LRM) as well as declarations of types and objects.

User-program statement identifiers and scope identifiers are expressed in DBUG commands using a simple extension to Ada name qualification. Statements are referenced by suffixing their procedure name by a dot followed by the sequential statement number relative to the start of the procedure. This is the same statement number used by the Unit Lister. When the user wants to put a breakpoint at a statement in the current scope, no subprogram name is necessary. To distinguish between spec and body, the statement number may be preceeded by a letter indicating either spec (S) or body (B). Body is the default.

A subprogram is referred to by name, unless it is overloaded. In that case it is referred to by its name followed by a specification of its parameter types. The parameter specification is similar to the formal part of a subprogram specification (Ada LRM 6-1), but without default expressions. It may be abbreviated by leaving out identifier lists and modes or entire parameter specifications which are not needed for overload resolution. For functions, the parameter specification may optionally be followed by the return type. DBUG will issue an error message if a subprogram specification is ambiguous.

Examples:

```
BREAK BEFORE 5                          -- sets breakpoint before statement
                                        -- 5 of current scope

BREAK BEFORE proc(x,y: in integer; z: out real).B10
                                        -- sets breakpoint before statement
                                        -- 10 of a definition
                                        -- of proc in the current scope
BREAK BEFORE proc(integer; integer; real).10
                                        -- refers to the same statement
```

```
BREAK BEFORE proc(z: out real).10
                                    -- proc is identified by its third
                                    -- parameter
BREAK BEFORE proc(integer).10       -- proc is identified by its first
                                    -- parameter

BREAK BEFORE func(integer)real.10
                                    -- sets breakpoint before statement
                                    -- 10 of a definition of the
                                    -- function func in the current
                                    -- scope

SCOPE TO prog.proc                  -- change scope to inside
                                    -- procedure "proc" inside scope
                                    -- "prog"
```

3.3.1.2 Command Repertoire

The following command descriptions use the following notation conventions: square brackets [] surround options; parentheses () surround optional words that help convey the meaning of the command; angle brackets <> surround higher-level constructs.

(a) Breakpoint Commands. Breakpoint commands establish breakpoints in the user program. No other action is taken. A breakpoint represents a place in a program where execution is to be suspended to allow the user to examine various aspects of the program. User commands to DBUG can be seen as executing at the point that the user program is halted.

Breakpoints can be established at statements and the entry or exit of subprograms, or on the occurrence of exceptions. The STEP command also defines a breakpoint. Breakpoints may not be permitted at all statements in optimized code. See Section 3.2.4.2.

Following is a summary of the breakpoint commands. Each keyword given below may be prefixed by a breakpoint identifier and/or the keyword BREAK.

```
AFTER <list of statement and label identifiers>

                                    -- break after the specified
                                    -- statements and labels

BEFORE <list of statement and label identifiers>

                                    -- break before the specified
                                    -- statements and labels

ON_STEP                             -- perform this procedure at the
                                    -- completion of single step
```

```
MODIFY (OF) <list of variables>

                              -- break before each statement
                              -- that modifies the specified
                              -- variables

EXCEPTIONS <list of exceptions>

                              -- break on raise of these
                              -- exceptions

ALL_EXCEPTIONS                -- break on raise of all
                              -- exceptions

UNHANDLED_EXCEPTIONS          -- break on unhandled exceptions

ON_ENTRY                      -- break on entry to all
                              -- subprograms and entries

ON_EXIT                       -- break on exit from all
                              -- subprograms and entries
```

Following are the breakpoint modification commands.

```
DEACTIVATE <list of breakpoint
            identifiers>|ALL|<breakpoint command>

                          -- suspends action of breakpoint

REACTIVATE <list of breakpoint
            identifiers>|ALL|<breakpoint command>

                          -- restores action of breakpoint

REMOVE <list of breakpoint statement and/or label
        identifiers>|ALL|<breakpoint command>

                          -- removes and  forgets the
                          -- specified breakpoints.
```

Commands that set breakpoints may be labelled by an identifier followed by a colon. Those commands not labelled by the user are given a unique name by DBUG. The unique names are constructed by concatenating the letters "bkpt" with an integer string. (Examples: bkpt1, bkpt465.)

The breakpoints set by the commands can later be referred to by breakpoint identifier. If the user specifies a breakpoint command with an identifier which is already in use, the command is appended to the commands already associated with that breakpoint identifier.

Breakpoint commands may contain a sequence of DBUG commands. These commands are executed when the breakpoint is encountered in the flow of the user program. After the commands have been

executed, control is normally returned to the user. However, if one of the commands is an execution control command such as PROCEED, control will be returned to the program at that point. If no sequence of commands is specified, the program execution is halted and the user is allowed to give DBUG commands interactively.

These stored commands are specified by the keyword BEGIN following the breakpoint command on the same line. The stored commands may then be typed, one per line or separated by semicolons as in the MAPSE Command Language. The BEGIN is terminated by the matching occurrence of the keyword END. A stored command sequence can contain nested BEGIN-END blocks.

Breakpoints can be removed by use of the REMOVE command. The breakpoint and its stored actions can be suspended by use of the DEACTIVATE command. The REACTIVATE command restores the actions of a breakpoint that has been DEACTIVATEd. These three commands can be followed by a breakpoint identifier list. Thus entire groups of identifiers can be manipulated. The keyword ALL is permitted as a breakpoint identifier and specifies all breakpoints in the user program.

A second form of the REMOVE, DEACTIVATE and REACTIVATE commands is supplied to permit finer control over breakpoint insertion and deletion. Each of the breakpoint commands may be preceded by any of the three keywords of the breakpoint modification commands. For example:

```
REMOVE BEFORE CALC_SINE.24

DEACTIVATE MODIFY MASTER_SWITCH
```

Only the specified breakpoints are affected. These breakpoint commands following the modification keyword may not include the list of stored commands. (No BEGIN keyword is permitted at the end of the breakpoint modification command.)

The stored commands for a breakpoint provide the DBUG user with conditional breakpoints. For example:

```
*
* BEFORE STMT 26 BEGIN
  2/ %M_S := MASTER_SWITCH
  3/ IF %M_S = ON THEN
  4/      PROCEED
  5/ ENDIF
```

This breakpoint command sets a breakpoint at statement 26 of the current scope. When that statement is encountered, the value of the program variable MASTER_SWITCH is fetched and stored in the DCP

variable %M_S. The DCP variable is tested and, if found equal to the string "ON", causes the breakpoint to return control to the executing user program. Any other value of %M_S causes control to be automatically returned to the DBUG command processor. See the next section for an explanation of the PROCEED command.

Breakpoints installed in task bodies cause the breakpoint to occur at the specified point for all instances of the task. This implies that the breakpoint occurs in the next task instance that encounters the breakpoint. To have a breakpoint for a specific instance of a task, the DBUG user can use the conditional breakpoint commands and the assignment command to fetch information from the program and check for the correct instance of the task.

(b) Execution Control Commands. DBUG supports the following commands to control execution of the user program. Most commands are executed at a breakpoint and have an immediate effect. They cause control to immediately return to the user program in the manner specified, thus closing the breakpoint. The exceptions are the ABORT, PRIORITY and DELAY which do not result in execution resuming and have effect only when the program is resumed. Control only returns to the DBUG Command Processor when another breakpoint in the user program is encountered, or when the user hits the "interrupt" key. The commands and their effects are given below.

GOTO <statement or label identifier>	proceed from the specified statement or label
PGM_RETURN [<expression>]	return from subprogram. <expression> is need for function return values.
RAISE [<exception id>]	raise exception and proceed
CALL <subprogram call>	invokes a user program which must be in the load module. Parameters are TBD.
DELAY <task identifier> <n>	delay a task for n seconds
PRIORITY <task identifier> <n>	change the priority of a task to n
ABORT <task identifier>	abort the specified task
STEP [<n>]	stop after Nth executed stmt beyond current stmt
PROCEED	continue the user program at the point where it was suspended.
IGNORE <n>	ignore breakpoint next n times

The integer argument to the IGNORE command specifies the number of times to suspend the current breakpoint. Control is not returned to the DBUG Command Processor from the current breakpoint until control has passed it the specified number of times. An argument of one (1) means skip the breakpoint once, and halt on the second occurrence.

The RAISE command takes an optional argument, like the Ada language statement. It is the exception to be raised. When no exception is specified, the raise handler re-raises the current exception. When there is no current exception, no processing is done. The syntax for the exception name is the same as in the Ada LRM.

The number of statements to the STEP command is optional, and is defaulted to one.

The GOTO command has the same effect as the Ada goto statement; it causes program control to be transferred to the specified label or statement. The only check on the GOTO command is that the label or statement is in the subprogram or (package) enclosing the current breakpoint. The user is permitted to specify a GOTO that is illegal by Ada language rules. These commands are not available unless the DEBUG (ALTER) option is specified at compile time.

The PGM_RETURN command has the same effect as the Ada return statement; it causes the subprogram enclosing the current breakpoint to terminate. If the subprogram is a function, the PGM_RETURN command must be followed by an expression which becomes the return value of the function. The types of function return parameters which are allowed are a subset of the parameter types allowed for the DBUG CALL command.

(c) DBUG Control Commands. DBUG Control Commands, listed below, are directives that control the execution of DBUG itself.

VERBOSE [ON | OFF]	default is on
BASE [<n>]	set default base for output of variables
TRACE [ON | OFF] [option] [<n>]	sets trace option on or off, default is on. Options are chain, flow, task, all. n specifies the circular buffer size
APPEND [OFF] [[ON] <file> [ONLY]]	a copy of breakpoint and display information is appended to the given file. There is no output to terminal when ONLY is specified

SCOPE CALLER	change scope to caller of current scope
SCOPE ENCLOSING	change scope to static enclosing scope
SCOPE RESET	reset scope to original breakpoint scope
SAVE <file>	save current breakpoints in file
LOAD <file>	load saved breakpoints and DCP variables from file
EDIT <breakpoint id>	invoke the editor on the commands associated with this id
PERFORM <file> [<parms>]	executes file as a stream of DBUG commands
<interrupt>	stop the user program from execution, and return control to DBUG
<new name> RENAMES <old name>	allows the renaming of commands and variables e.g., trap renames break %aileron renames main.controls.aileron
CP [<command> [-&]]	invokes a command processor and processes the command if present
RETURN [SAVE <filename>]	return to caller (e.g., MCP). The save option saves current breakpoints and DCP variables to file

The VERBOSE command affects the length of the prompt given the user when the executing program reaches a breakpoint; OFF makes a shorter prompt.

The TRACE command tells DBUG to keep a circular buffer for the option specified. The chain buffer contains currently active recent subprogram calls. The flow buffer contains recent flow of control. The task buffer contains a trace of recent tasking events. These buffers may be inspected by the use of the special variable %TRACE.

The APPEND command redirects output for all subsequent DBUG actions. If the file did not previously exist, it is created. The ONLY option of the output command causes no output to the user's terminal other than a brief report when a breakpoint is reached.

The SCOPE commands change only the visibility of identifiers for the DBUG commands. Each statement is contained within some scope in an Ada program, and DBUG preserves this viewpoint for the user. This means that all variables visible in the current scope where the program is halted are also visible to the user. (At a breakpoint within a declarative part, only declarations which have been elaborated are visible.) Other variables in other scopes are available via name qualification.

The SAVE command outputs a text file containing the necessary breakpoint commands and DCP variable values to recreate the current state of DBUG's data base. Only the breakpoints and variables are recorded on this file, not current execution state, scope, etc.

To read in a file containing SAVEd breakpoints and variables, the LOAD command is used.

The EDIT command allows the user to edit previously specified breakpoint scripts.

The PERFORM command allows the user to execute a script of DBUG commands which are in a file.

The user terminates a DBUG session by using the RETURN command. The SAVE option may be used to save the current breakpoints to the specified file.

(d) Information Commands. Information Commands permit the user to examine the state of the user program.

PUT <expression> [BASE <n>] [VERBOSE]	to print out several variables, <expression> can be an aggregate. Put with the verbose option prints each variable's name with its value
PUT_SCOPE [BASE <n>]	displays values of all variables in scope
DISPLAY TASK [<options>] [<n>]	display the status of tasks at n levels, starting at the current block

<options>:

ACTIVATED	tasks currently activated
RUNNING	tasks that are running

CALL [entry]	all tasks that are waiting at any call point or those tasks that are waiting at a call point for a particular entry [entry]
ACCEPT [entry]	all tasks that are waiting at any accept point or those waiting for a particular entry call
DELAYED	tasks that are waiting because of a programmed delay
DEPENDENT	tasks which are dependent on the current block
BLOCKED	tasks which are blocked and why
ALL	all options
WHAT BREAK [<bkpt id>]	default is to display all break-points
WHAT SCRIPT <bkpt id>	displays command and script associated with bkpt i.d.
LIST <options> [<n>]	list source text for n lines. If n is negative, list n lines above and below current line
<options>:	
<stmt id> | <label id> | *	list at the specified stmt (* means current stmt)
[d][r][m] [<variable>]	list at lines containing references to the variable according to the xref options specified (d-declared, r-referenced, m-modify). No variable name means use the variable specified in the last list command, and show the next occurrence

The PUT_SCOPE command is a shorthand for displaying all the variables in the current scope. The user can redirect the output of this command by specifying a file name (ex. PUT_SCOPE -> prog.dump).

The base option of the PUT and PUT_SCOPE commands is an integer from two (2) through sixteen (16) specifying the base of the numeric type desired and applies to the display of all variables in the variable list. When no base is supplied, the variable is printed

out in its own mode (string, integer, floating point, enumeral, etc). When the variable is a composite object (array, record, etc) each component is printed in its own mode, unless a base was specified. In that case, all of the components of the variable are printed in the specified base.

(e) Predefined Variables. DBUG has the following predefined variables:

%HI_PRIORITY	all tasks with priority greater than this will continue executing when control is returned to DBUG
%SCOPE	name of current scope
%VARIABLES	an array of all variable names defined in the current scope
%TRACE.NEST(1..n)	shows n currently active subprograms and their parameter lists
%TRACE.CHAIN(1..n)	shows recently called subprograms
%TRACE.FLOW(1..n)	shows recent flow of control
%TRACE.TASK(1..n)	displays trace of tasking events
%ENVIRONMENT	the same as the MCP predefined variable
%STATUS.CP	status returned from the cp command

The %TRACE variable is maintained by DBUG as a record of arrays of records. The trace variables, with the exception of %TRACE.NEST are maintained at the user's request (see the TRACE command) with the number of records to be maintained being defined at the time of the request. The %TRACE.FLOW variable is not available if the DEBUG(BREAK) parameter was not specified at compiler time.

f) Comparisons between MCL and DCL. The following commands and predefined functions are common to both MCL and DCL:

BEGIN, END	
WHILE	
FOR	
IF	
CASE	
->	
-<	
GET	
:= (assignment operator)	user-program variables may also be assigned to in DCL
PROCEDURE	
EXEC	
USE <DCP or user_program variable>	one of each variable is allowed at a time

```
DUMP_VARS
HELP                              DCL provides help for debug only

CONTENT()
BOOLEAN(), REAL(), INTEGER(), STRING()

The following MCL commands are not present in DCL:

CREATE_SIMPLE
CREATE_COMPOSITE
DELETE
COPY
RENAME

ABORT
WAIT
START
CANCEL
STATUS

LOGOUT
SUSPEND                    the DCL SAVE command is very similar
-:
-&
```

3.3.1.3 Processing

COMMAND PROCESSOR is structured along the lines of the Mapse Command Processor. It is composed of the following major procedures. Driver reads user commands from a terminal or background script. It then calls Lexparse to parse the input, and passes a parse tree on to the Tree Interpret procedure to be interpreted. Tree Interpret uses Expression Processor to evaluate DCL expressions and the Variable procedure to manage DCP variables.

(a) Driver. The Driver loops to process commands sequentially. It processes each command by:

(1) invoking Lexparse to parse the next command in the input stream into a parse tree;

(2) invoking Tree Interpret to interpret the parse tree produced by Lexparse.

(b) Tree Interpret. The parse tree interpreter (Tree Interpret) borrows its design from MCP. It differs in that only a single execution of Tree Interpret is active at any time in DBUG, although several invocations of Tree Interpret can be stacked. This permits it to be implemented as a procedure.

Tree Interpret is called by Driver with parsed commands from a terminal or script, by Execution Control with a breakpoint script parse tree, or recursively for commands such as PERFORM and IF.

Tree Interpret takes as input: (1) a stream to be used as standard input for executing commands; (2) a stream to be used as standard output for executing commands; (3) a DIANA-like parse tree (built by Lexparse) to be interpreted; and (4) a subtree containing any actual parameters.

For constructs found in MCL, processing is the same as performed by MCP Tree Interpret. These include program invocation, assignment, expression, EXEC, IF, CASE, loops, EXIT, and blocks. DBUG control commands are also performed within Tree Interpret.

The DBUG-specific commands such as BEFORE, REMOVE, PROCEED, and DISPLAY TASK are implemented by calls to EXECUTION CONTROL PROCEDURES, BREAKPOINT COMMAND PROCEDURES, or INFORMATION COMMAND PROCEDURES.

For the PERFORM command, the file to be interpreted is parsed by Lexparse. Tree Interpret is recursively called with the resulting tree and the parameters to PERFORM.

For the SAVE command, calls are made to INFORMATION COMMAND PROCEDURES to convert all of the breakpoint script parse trees into character representation. These scripts and other state information are then formatted and written to a database object.

For the EDIT command, the DBUG Database is called to convert a breakpoint script parse tree into character representation and store it in a KAPSE database object. The editor is then invoked on this object. When the editor returns, Lexparse reparses the edited breakpoint script, and the script is given back to the DBUG Database.

(c) Expression Processor. Expression Processor is similar to that of the MCP, being tailored to DBUG expressions. In addition to the MCP constructs, the Expression Processor also must recognize user-program variable names.

The Expression Processor evaluates parse trees by returning actual values of primitive (leaf) nodes (e.g., literals, variables), calling itself recursively on children of operator or function nodes, and then calculating the result of the operation or function with the returned values of the children.

(d) Variable. The Variable procedure maintains the DBUG "%" variables in an internal variable space. It is responsible for allocation, alteration, and assignment to and from variables. Variable implements scopes to support DBUG script parameters. Variable also detects occurrences of DBUG variables, calling the appropriate routine to compute or store the predefined variable's value.

(e) Script. Script is called by Tree Interpret to process a DBUG script header. It is passed the actual parameters that were passed to Tree Interpret. Script performs the same association of formal to actual parameters that MCP SCRIPT does.

(f) Program Invocation. Program Invocation implements the CP command by calling executable programs or scripts located in the KAPSE database. Its processing is the same as the MCP Program Invocation.

3.3.1.4 Outputs

COMMAND PROCESSOR calls BREAKPOINT COMMAND PROCEDURES, EXECUTION CONTROL PROCEDURES and INFORMATION COMMAND PROCEDURES with an indication of the command to be invoked and a parse argument list. For breakpoint commands, the outputs include the breakpoint script parse tree.

3.3.2 BREAKPOINT COMMAND PROCEDURES

BREAKPOINT COMMAND PROCEDURES handle the processes which create and maintain breakpoints and the breakpoint table.

3.3.2.1 Inputs

BREAKPOINT COMMAND PROCEDURES has two types of input. One is the necessary information to create a breakpoint, the other is the information to maintain the breakpoints and the breakpoint table.

To create a breakpoint, the processor accepts as input from the DCP the breakpoint identifier (if available), the option specified, the script locator (if a script has been specified), and the command line. Some commands require additional information. The BEFORE and AFTER commands require statement identifiers. The MODIFY command requires variable identifiers. The EXCEPTIONS command requires exception identifiers. The ALL_EXCEPTIONS, UNHANDLED_EXCEPTIONS, ON_STEP, ON_ENTRY, and ON_EXIT commands do not require any additional information.

The statement identifier that is provided for some breakpoint commands may include scope identification as well as the number of the statement. To maintain breakpoints, the commands DEACTIVATE, REACTIVATE, and REMOVE require breakpoint or statement identifiers.

3.3.2.2 Processing

The processing of breakpoint commands can be divided into two parts: one which creates the various breakpoints and the breakpoint table, and one which maintains the status of the breakpoints and the breakpoint table.

3.3.2.2.1 Creating Breakpoints

When COMMAND PROCESSOR receives a command to create a breakpoint, it passes the necessary inputs to the breakpoint processor. Each breakpoint command creates one of two kinds of breakpoint: statement breakpoints, which are associated with a specific statement, and global breakpoints, which are associated with an event such as an exception.

3.3.2.2.1.1 Statement Breakpoints

For the BEFORE and AFTER commands, PROGRAM LIBRARY ACCESS PROCEDURES (PLA) is called to get the statement table for the specified procedure. From this, the address for the hook (or first instruction if no hooks are present) for each statement specified can be determined. This address is stored in the breakpoint table with the breakpoint identifier (either user supplied or assigned), the option, the script locator (if one exists), and the command line string. If there are no hooks present, the sum of the lengths of those instructions which would have to be moved to place a hook is also recorded in the breakpoint table. If the AFTER option is specified for a statement in a procedure which has been compiled without hooks, an error message is issued, and no breakpoint is created.

To find the address associated with the specified label, PLA is called to determine which statement is marked by the label.

For the MODIFY command, PLA is called to check the cross reference for the statement identifier for each case where the variable is modified.

Once the above processing has been done, if hooks are available, the DSR is called to activate the hook in this location, and the activate flag in the breakpoint table is turned on. If no hooks are present, the instruction lengths is sent to the DSR to get those instructions which have to be moved. These are stored in the breakpoint table and the DSR puts an activated hook in that location. The activate flag in the breakpoint table is turned on.

3.3.2.2.1.2 Global Breakpoints

When the commands EXCEPTIONS, ALL_EXCEPTIONS, UNHANDLED_EXCEPTIONS, ON_STEP, ON_ENTRY and ON_EXIT are specified, the breakpoint identifier (either user supplied or assigned), the command, the script locator (if one exists), and the command line are stored in the breakpoint table, and the activate flag is turned on. When the EXCEPTIONS command is specified, the exception command identifier is recorded in the breakpoint table and sent to the DSR. When an ON_ENTRY or ON_EXIT command is specified, the DSR is informed to break whenever a subprogram is invoked or exited (depending on the option specified).

3.3.2.2.2 Maintaining Breakpoints

The DEACTIVATE, REACTIVATE, and REMOVE commands are used to maintain breakpoint status. Again there are different processes for hooked and unhooked code.

DEACTIVATE may be specified with either a list of breakpoint identifiers or statement identifiers as operands. For each statement identifier, the specific record is found in the breakpoint table, and the activate flag is turned off. If hooks are present, the KAPSE is called to deactivate the hook. If no hooks are present, the DSR is passed the code to be restored. For each breakpoint identifier, the processor reads through the breakpoint table and the records with that identifier are processed as above.

When REACTIVATE is specified, the processing is the same as for DEACTIVATE, except that the activate flag is turned on, and the KAPSE is called to put an activated hook into the location.

When REMOVE is specified, the records are found in the table. The DSR is called to remove the breakpoint, with the instructions to be restored if no hooks are present.

3.3.2.3 Outputs

The outputs of BREAKPOINT COMMAND PROCEDURES include the breakpoint table, hooks to be inserted into code, instructions that were overlaid by breakpoint hooks, indications to DSR that exceptions, program entry and exit were called for, and an indication as to whether or not a breakpoint is active.

3.3.3 EXECUTION CONTROL PROCEDURES

EXECUTION CONTROL PROCEDURES processes DBUG commands that affect the execution of the user program. These commands resume execution of the user program and control tasking. It also initiates processing when control is passed to DBUG at a breakpoint.

3.3.3.1 Inputs

The inputs required by EXECUTION CONTROL PROCEDURES with the different commands received from COMMAND PROCESSOR. The IGNORE command requires the current breakpoint and a parameter. The STEP command requires an integer parameter. The GOTO command requires a statement or label identifier. The PGM_RETURN COMMAND may require a return expression. The CALL command requires the subprogram name. The task control commands require the task name and a parameter in the case of DELAY and PRIORITY.

At a breakpoint, the current address and the type of break are required. In the case of a break at exception, the exception identifier is also required.

3.3.3.2 Processing

3.3.3.2.1 Resuming a User Program

The following processing is done when a command is issued to resume the execution of the user program.

If IGNORE is specified, the count is stored in the breakpoint table and a command to proceed is issued. Then when the current breakpoint is reached again, EXECUTION CONTROL PROCEDURES will check to see if this is the nth time. If not, a command to proceed will be issued. If so, the processing for the breakpoint continues as necessary. When a STEP command is issued, UTILITY PROCEDURES is called with the STEP count. The STEP command is not available for unhooked code.

When a GOTO is issued, if a statement identifier has been specified, PROGRAM LIBRARY ACCESS PROCEDURES is called to get the statement table for the correct subprogram. From this the address for that statement is determined. This address is passed to UTILITY PROCEDURES with a command to proceed. If a label identifier is specified, the PLA is called to determine which statement is associated with that label. The statement is then processed as above.

When a PROCEED command is issued, UTILITY PROCEDURES is passed the current program address and commanded to proceed. If the current address has an instruction save for it in the breakpoint table, it is also passed to UTILITY PROCEDURES to be executed. When a PGM_RETURN is issued, if an expression is included this will be evaluated. The PLA is called to determine the epilogue address. UTILITY PROCEDURES is called to assign the return parameter, and to process a proceed at the address.

When a CALL command is issued, UTILITY PROCEDURES is used to initialize the IN formal parameters. The Program Library is then accessed to get the address of the subprogram, and the call is made through UTILITY PROCEDURES. When the subprogram returns, control returns to COMMAND PROCESSOR and the OUT parameters are assigned to DBUG variables.

When a RAISE command is issued, the exception identifier is looked up in the Program Library and sent to UTILITY PROCEDURES with an instruction to raise the exception.

3.3.2.2.2 Task Control

When a DELAY, PRIORITY, or ABORT command is issued, the task identifier is determined. This task identifier, the action to be taken (change priority, abort or delay) and, in the case of PRIOIRTY or DELAY, an integer specifying the new priority or the delay value respectively, is passed to UTILITY PROCEDURES.

3.3.3.2.3 Returning From a User Program

The following processing is done when control is passed to DBUG at a breakpoint.

An exception breakpoint takes precedence over any other type. When an exception breakpoint is reached, the breakpoint table is checked for an entry which contains the encountered exception identifier. If one is found, a message is issued and control is passed to COMMAND PROCESSOR along with any script that may exist for that breakpoint. If not, the table is checked for an ALL_EXCEPTIONS entry. If that is found, a message is issued and the control is passed to the Command Processor with any existing script.

In the case of an unhandled exception, the breakpoint table is checked for an associated script. If found, a message is issued and this script is associated script passed to the COMMAND PROCESSOR. If not, a message is issued and control returns to the user.

When an activated statement hook is reached, the current address is used to find the corresponding record in the breakpoint table. The ignore count is checked and if zero, a message is issued and control is passed to the Command Processor with any script that may exist for that breakpoint. If not, a command to proceed is sent to UTILITY PROCEDURES. If the option for the breakpoint is AFTER, UTILITY PROCEDURES is told to step one hook. When control returns, a message is issued, and control is passed with a script (if one exists) to the COMMAND PROCESSOR.

When an ENTRY or EXIT breakpoint is reached, the breakpoint table is checked for a script. A message is issued and control is passed to the Command Processor with the script if it exists.

3.3.3.3 Outputs

The outputs from EXECUTION CONTROL PROCEDURES include the address for proceeding, the exception identifier for a RAISE, task identifier and parameter for task control, and message information and script locators for breakpoints.

3.3.4 UTILITY PROCEDURES

UTILITY PROCEDURES consists of several subprograms and tasks that provide functions used by other DBUG CPCs. Some of these subprograms are called directly from the other CPCs and provide communication with the KAPSE facilities, including the KAPSE.MULTPROG and KAPSE.RTS. All machine-dependent and KAPSE-specific processing is contained within the Utility Procedures.

3.3.4.1 Processing

3.3.4.1.1 DBUG Support Routine

The DBUG Support Routine (DSR) will process all information and commands from the user program and KAPSE.RTS to DBUG. This includes the processing of breakpoints when they are encountered and interpreting messages from the KAPSE.RTS when exceptions have been raised or when history events (tasking, flow, program entry and exit) have occurred. It will also process information and commands sent by DBUG to the RTS of the user program. These include tasking control commands, accessing of program values, and the manipulation of breakpoints.

When a breakpoint is encountered or, in single step mode, before the beginning of each statement is executed, control passes from the user program to the DSR. The DSR will determine the address of the statement and will pass this address along with an indication of whether the break occurred because of an active breakpoint or because of single stepping to the DSR interface procedures.

When the KAPSE.RTS encounters an exception, it will make a call to the DSR indicating what exception has taken place. The DSR will determine the address at which the exception occurred and, if necessary, will return to UTILITY PROCEDURES, passing the exception identifier and this address.

If any TRACE command has been issued, the KAPSE.RTS will return to the DSR whenever an event specified by the command takes place. It will return with an indication of the type of event (task, flow, procedure entry or exit), an indication of the sub-type event (e.g., for tasking, the sub-types include activation, entry call, delay, begin and end rendezvous, etc.), and the address where the event occurred. The DSR will enter this information into the appropriate circular buffer for later access by INFORMATION COMMAND PROCEDURES.

The DSR is also called to continue user program execution. It is passed the address where execution is to resume and it instructs the KAPSE.RTS to proceed. This function will be invoked not only to proceed immediately from a breakpoint, but also as a result of the GOTO, PGM_RETURN and CALL instructions.

If the program has been suspended at a breakpoint which had overlaid an instruction (i.e., a breakpoint is placed at a statement for which there was no hook), the DSR will supply the overlaid instruction to KAPSE.RTS to be executed before resuming execution beyond the breakpoint.

3.3.4.1.2 DSR Interface Routine

The KAPSE provides for communication between DBUG and the DSR which is linked to the runtime system of the user program. In order to localize KAPSE-dependent processing, the DSR interface routine will process the communication between the DBUG CPC's and the user program. This routine will translate the commands and request for data into interprogram communication primitives to be transferred to the DSR and will transform the data received from the DSR into a format expected by the calling CPC.

3.3.4.1.3 Ada Expression Evaluation

(a) Ada Parse Procedure. The Ada Parse procedure parses the subset of legal Ada expressions and Ada name references that are supported by DBUG. It is called by the Expression Processor and other procedures to parse variable names, scope identifiers, labels, etc. The output is an abstract syntax tree that can, by a call to the Analyze procedure, be turned into a legal DIANA tree.

(b) Evaluate Procedure. The Evaluate procedure is used to reference and modify the variables, parameters and other data items of the program being debugged. It is called when processing the assignment, PUT and CALL commands, and to examine the program stack (eg. for %TRACE.NEST).

Evaluate accepts a DIANA tree that represents the expression. It also accepts or generate a DCP literal value. Evaluate accesses the user-program data via calls to the DSR.

Evaluate interprets the storage information in the DIANA tree to access the user-program data item. It also uses the storage and type information to convert between the machine representation of the data item and the DCP representation.

3.3.4.2 Outputs

The output of the procedures in UTILITY PROCEDURES take the form of commands and data sent to the KAPSE.RTS and of data sent to other DBUG CPCs. This output is specified in the description of these procedures above.

3.3.5 INFORMATION COMMAND PROCEDURES

INFORMATION COMMAND PROCEDURES performs the processing necessary for the DBUG Information commands. This includes displaying variables of different types (e.g., records, enumerals, etc.) in a readable format, as well as displaying DBUG data such as the output of trace functions and breakpoint scripts. These procedures also format the program listing when requested by the user.

3.3.5.1 Inputs

INFORMATION COMMAND PROCEDURES is called by COMMAND PROCESSOR any time the user requests information to be displayed. The input is the identification of that information. For displaying user variables the identification will include the variable name and scope. For the display of trace information, COMMAND PROCESSOR will indicate the type of information to be displayed (flow, task, etc.). For listing the program source, the inputs will include an indication of the starting line number and the number of lines. When the user requests that a script be listed, the command processor will supply the breakpoint identifier for the script. These procedures are called from EXECUTION CONTROL PROCEDURES to identify breakpoints and exceptions to the user when encountered. EXECUTION CONTROL PROCEDURES will supply the current address and scope information and the exception identifier in the case of an exception.

In addition to the commands sent by COMMAND PROCESSOR and EXECUTION CONTROL PROCEDURES, other inputs will include the breakpoint table that is maintained by BREAKPOINT COMMAND PROCEDURES and the program source when the listing function is invoked.

3.3.5.2 Processing

The processing performed by INFORMATION COMMAND PROCEDURES varies according to the information that is to be displayed.

For the PUT command, this CPC calls the Expression Processor to evaluate the input argument. The result is then formatted for display to the user. For the PUT_SCOPE command, the Program Library is accessed to find all the variables in the current scope. The value of each variable is then found with the Evaluate procedure and formatted for display to the user.

For displaying trace information, the package is passed the type of information to be displayed. In the case of FLOW and CHAIN, INFORMATION COMMAND PROCEDURES will call UTILITY PROCEDURES requesting the information that has been maintained in the circular buffers.

For NEST, a call will be made to UTILITY PROCEDURES to access the user program's stack for the identity of the subprogram in the current call chain. PROGRAM LIBRARY ACCESS PROCEDURES will then be called to obtain the addressing information for the parameters of these procedures. UTILITY PROCEDURES is called with these addresses to get the values of these parameters. When tasking information is requested, UTILITY PROCEDURES is called to access information maintained by the KAPSE. RTS regarding the currently active tasks and their status.

When a request is made to list lines from the program source, a call is made to PROGRAM LIBRARY ACCESS PROCEDURES to supply the specified lines and these are formatted and displayed.

To display the list of current breakpoints or the script associated with a particular breakpoint, INFORMATION COMMAND PROCEDURES will gather the information from the breakpoint table in the DBUG database. In displaying scripts, the parse trees must be converted to a textual form.

When a breakpoint is reached, EXECUTION CONTROL PROCEDURES will call INFORMATION COMMAND PROCEDURES indicating which breakpoint has been reached and this package will format a message including scope, task and nesting information.

3.3.5.3 Outputs

The output of this package is the specified information displayed on the output device.

3.3.6 PROGRAM LIBRARY ACCESS PROCEDURES

PROGRAM LIBRARY ACCESS PROCEDURES provide DBUG with access to information generated by the Compiler and Linker which has been stored in the Program Library of the executing program. These packages provide identification of user-program names, information regarding the addressing of variables, identification of the tasks and sub-programs within the running program and source listings.

3.3.6.1 Inputs

The inputs to this CPC differ according to the information that is being requested and are described below for each.

3.3.6.2 Processing

(a) Analyze. The Analyze procedure is called to look up all user-program names recognized by DBUG. These names include program variables, exception names, and formal parameter names found in the cross reference.

The inputs include the name, the program environment, which specifies a point in the user program, and a DIANA tree for an Ada expression. Analyze references the complete program symbol table to look up the name.

Processing is similar to the semantic analysis phase of the Compiler Front End (COMP.FE), using the Names and Expression Package from the compiler front end. The output of Analyze is a completed DIANA tree contained addressing information.

(b) Statement Table Access. The statement tables generated by the compiler are used to re-locate the address of a statement number or to identify a statement number when its address is known. They are used to set breakpoints, process single step, implement the GOTO commands, and display trace and task information.

The Statement Table Access procedures accesses these statement tables to return the requested information. It also uses the Relocation Map to convert between relative and absolute addresses.

(c) Relocation Map Access. The relocation map refers to all of the symbol and statement tables which make up a program. It is used by Statement Table Access and Evaluate to translate between an address relative to a compilation unit and an address global to the entire user program. The map is also used to determine the new scope when the user changes current scope.

Information in the relocation map can be accessed by either a global addressed or the name of the relocatable unit. The names of all compilation units in a program can also be requested.

(d) Cross Reference Access. DBUG inspects the global cross reference for the locations at which a user-program name is either declared or referenced, or where a variable is modified. This information is needed by the LIST command. The MODIFY breakpoint command only requires locations at which variables are modified.

The input to Cross Reference Access is a name which has been processed by Analyze and the types of references which are to be returned. The name can be a variable name or any other type of name kept in the cross-reference data base (e.g., a type name).

The package is implemented using the CROSS REFERENCE PACKAGE in PIF.PLIF.

(e) Source Pretty Printer. Source Pretty Printer is used to implement the LIST command. For the range of lines specified, it returns a formatted version of the source program, including comments. This package is implemented using the Source Reconstruction Package in PIF.PLIF.

3.3.6.3 Outputs

The outputs to these procedures differ according to the information that is being requested and are described above for each.

3.3.7 DBUG DATABASE

DBUG DATABASE controls global DBUG information which must be accessed by more than one CPC. This information includes the breakpoint tables, the breakpoint scripts and the current program state.

3.3.7.1 Inputs and Outputs

The breakpoint tables are maintained by BREAKPOINT COMMAND PROCEDURES and EXECUTION CONTROL PROCEDURES. Associated with each breakpoint identifier are one or more groups of breakpoints, grouped according to the commands such as BEFORE and MODIFY which set them. For the breakpoint identifier there is also its activated flag and a locator to the breakpoint identifier's parse tree of stored commands.

Information is also kept for each statement which has a breakpoint. This includes an activated flag, the ignore count, the label by which the user referred to the statement, and invariant program state information such as the address and scope.

The breakpoint information may be located in many ways, such as by breakpoint identifier, statement identifier, label or address.

The information for the current program state includes the program counter, the stack pointer, descriptions of the scope and task, and the currently raised exception.

3.3.7.2 Processing

Processing involves insertions, deletions and searches of the database.

3.4 Adaptation

3.4.1 DBUG Size Restrictions

DBUG has no size restrictions other than those imposed by the KAPSE.

3.4.2 DBUG Extensions

DBUG is carefully designed to permit extensions to the set of commands for future AIE development. It is expected that APSEs will use the debugging facilities as a test bed for debugging embedded software applications by writing various control scripts and possibly extending its set of commands and control over the user program execution. The nature of these extensions is expected to be in the direction of environmental and functional simulation of the target environment.

LEFT BLANK INTENTIONALLY

4.0 QUALITY ASSURANCE PROVISIONS

4.1 Introduction

Development testing of DBUG will be conducted in three stages. The first stage is subprogram testing which tests each CPC and its subunits. The second stage is CPCI testing which will test the interaction between CPCs. The third level is subsystem integration testing which will verify the Debugger's performance in combination with other MAPSE tools (e.g., Compiler, Linker, etc.) that produce output on which it depends. The results of the development will be documented and submitted to Quality Assurance (QA).

Formal CPCI testing will be conducted on DBUG according to formal Test Procedures. Since the Debugger subsystem consists of a single CPCI there are additional formal subsystem tests for the subsystem DBUG. Acceptance testing of DBUG will be conducted according to the AIE Test Plan [AIE(1).TPLAN(1)] and Section 4.3.

4.2 Test Requirements

This section describes the requirements and techniques for development testing at each of the levels described above. It also lists the test requirements for formal CPCI testing.

4.2.1 Development Testing

4.2.1.1 Subprogram Testing

Each CPC internal to DBUG, as described in the Computer Program Product Specification, and each subunit contained within a CPC will be tested in the following manner. For each CPC, an input driver and a test output module will be developed. The input driver will accept input devised by the implementor to exercise each of the CPC's subunits. In addition to legal input which will test the normal functioning of the CPC, illegal input will be submitted to verify all error conditions. The test output module will contain test stubs for each of the external procedures and packages with which the CPC interfaces. These tests stubs will produce verifiable results (e.g., a human readable representation of a parse tree from the command processor) as well as data suitable for input to the input drivers of other CPSs. Each output test module will also provide suitable return values to its CPC. Test descriptions and results for each CPC will be submitted to QA.

4.2.1.2 CPCI Testing

CPCI testing will be performed to ensure the reliability of the interfaces between the CPCs. All interactions between the CPCs will be exercised and the validity of the communication verified. This verification will include the inspection of input and output parameters as well as the effect produced on shared data. In some cases these tests will utilize the data generated by the test output modules described above. Detailed descriptions of these tests and test reports will be submitted to QA.

4.2.1.3 Subsystem Integration Testing

Subsystem integration tests will be designed to verify proper interfaces with the other components of the AIE on which the Debugger depends. These test will ensure that external data (e.g., the statement table generated by COMP.BE) is correctly accessed and processed. This testing will also exercise the real time interaction with the KAPSE.RTS. Support may be needed from the other AIE components in the generation of test data.

4.2.2 Formal CPCI Testing

The following design requirements, which are specified in this document, will be verified by formal CPCI tests. The tests will be described in the Test Procedures [AIE(1).DBUG(1).DBUG(1).TPROC(1)].

(1) Use of the DBUG

a. DBUG is invoked with four parameters:

- The first specifies which program DBUG is to control - either the name of a program that is to begin execution, or the name of the program context of a suspended program.
- The second consists of an ASCII string containing the parameters to be passed to the program to be debugged; if DBUG is being called for a suspended program, this parameter is ignored.
- The third specifies the source from which DBUG commands are input; default is standard input, can contain the name of a script for background debugging.
- The fourth specifies where the output is to be directed; default is standard output, can be a file for background debugging. (3.2.4).

b. A suspension of a program executing under DBUG will cause the control to return to the DBUG Command Processor; a suspension is:

- The user hits an "interrupt" key. (3.2.4.2.2)
- An unhandled exception. (3.2.4.2.1.c)

c. After an unhandled exception during normal program execution, DBUG may be invoked to examine the state of the program at the point where the exception was raised. (3.2.4.2.1.c)

d. After a suspension due to an interrupt during normal program execution, DBUG may be invoked at the point that the interrupt occurred. (3.2.4.)

(2) Language Overview

a. A command consists of a keyword specifying an action and any required parameters.

b. Lists of identifiers provided as parameters are separated by commas; identifiers include variable names, statement identifiers, and exception names.

c. A command is terminated by a semicolon or a newline.

d. User-program variables can be expressed in normal Ada syntax;

- Available are access dereferencing, array subscripting, and record component selection.
- Expressions used as subscripts may not include function calls or overloaded operators.

e. DBUG Command Processor variables are analogous to MCP variables;

- Begin with %.
- Values can be set and retrieved.
- MCP variables from the processor where DBUG was invoked are not accessible.

f. Ambiguous names may be qualified using normal Ada syntax.

g. User-program statement and scope identifiers are specified using an extension to Ada name qualification;

- Within the current scope, use the sequential statement number relative to the start of the subprogram or package; the number is the same as that given the compilation unit.

- For bodies outside of the current scope, use the subprogram or package name followed by a dot and then the sequential statement number.

- To distinguish between spec and body, the statement number may be preceded by a letter indicating either spec (S) or body (B); body is the default.

- For overloaded procedures or functions, the name is followed by a distinguishing parameter specification and return type if necessary. (3.3.1.1.1)

(3) Compiler Interface

a. When a DBUG command might not have its intended effect due to the OPTIMIZE or DEBUG parameters or the pragma OPTIMIZE of the compilation unit, DBUG prints a warning message and then tries to process the command.

- Without DEBUG (BREAK), the AFTER, STEP, ON_STEP, and the FLOW option for the TRACE are not available.

- DBUG checks the options and the pragma OPTIMIZE of the compilation unit and determines the restrictions for the whole unit by the most optimized level found in the unit.

- Without DEBUG (ALTER) and with compiler parameter or pragma OPTIMIZE being TIME or SPACE, the Execution Control commands and the modification of program variables are not guaranteed to have their intended effects, and the GOTO function is not available. (3.2.4.1)

(4) DBUG Commands

The following requirements are divided into four sections covering the Breakpoint, Execution Control, DBUG Control, and Information Commands. For each section, the first requirements are the ones that are applicable to all or most of the commands. These are followed by the individual requirements for the commands of the section. This part will avoid repeating the basic functional requirements of each command since the appropriate location within this document can be referenced when tests are written to exercise each command. Requirements that will be specified are those which list defaults for parameters, maximum values, results of nteractions, and special conditions which are not readily discernible from the functional descriptions.

Breakpoint Commands

a. A breakpoint on an exception takes precedence over any other type of breakpoint. (3.3.3.2.3)

b. All breakpoints must have names.

- The user may specify the name by prefixing the breakpoint command with an identifier followed by a colon.
- If the user does not specify a name, a unqiue one will be generated by concatenating the string form of an integer with the letters "bkpt".
- The name serves as an abbreviation for all the breakpoints created by the breakpoint command.
- The identifier has the syntax of an Ada identifier. (3.3.1.2.a)

c. Breakpoint commands may contain a sequence of DBUG and DCP commands to be performed when the breakpoint is encountered.

- Stored commands are specified by the keyword BEGIN following the breakpoint command on the same line.
- Stored commands are typed one per line or separated by semicolons.
- Ended by a matching END.
- Can contain nested BEGIN-END blocks.
- After the commands are processed and if the last was not an Execution Control command causing execution to resume, the control will break to DCP. (3.3.1.2.a)

d. A user may specify a breakpoint command using a name already in use.

- The new breakpoints are appended to the old one already associated with that name.
- If a script of stored commands was specified with the old breakpoints, it will also be processed after a break on the new breakpoints.
- An issuance of a breakpoint command with stored commands will replace any old script which had been associated with the namee, and the new script will become associated with both new and old breakpoints. (3.3.1.2.a)

e. Breakpoints installed in task bodies cause the breakpoints to occur at the specified point for all instances of the task (3.3.1.2.a)

f. If an AFTER command is specified for a statement in a compilation unit without hooks, an error message is issued and no breakpoint is created. (3.3.2.2.1.1)

g. An exception raised will first be examined for a match with the EXCEPTIONS command, then the ALL_EXCEPTIONS command, and then if neither is found, the execution will resume with the normal exception handlers of the program (3.3.3.2.3)

h. The keyword ALL used as an argument to a breakpoint modification command specifies all the breakpoints in the user program (3.3.1.2.a)

i. Breakpoint modification commands cannot specify a script of stored commands. (3.3.1.2.a)

j. Finer control for specifying individual breakpoints can be accomplished by following the modification keyword with the breakpoint command specifying the desired breakpoint. (3.3.1.2.a)

Execution Control Commands

a. These commands are not guaranteed to have their intended effects if the unit was compiled without DEBUG => ALTER, and with OPTIMIZE => TIME (or SPACE) or with a pragma OPTIMIZE specifying one of the two. (3.2.4.1)

b. Statements or labels for the GOTO commands must be within the subprogram (or package) enclosing the breakpoint. (3.3.1.2.b)

c. They may specify transfers which are illegal by Ada rules. (3.3.1.2.b)

d. The RAISE command without an argument re-raises the currently raised exception or if there is none, just causes execution of the program to proceed. (3.3.1.2.b)

e. DELAY, PRIORITY, and ABORT affect tasks from the user-program but do not cause execution to proceed. (3.3.1.2.b)

f. The number argument to STEP is optional and defaults tc one. (3.3.1.2.b)

g. An argument of zero (0) to IGNORE means to skip the current breakpoints the next zero times (break on the next occurrence). (3.3.1.2.b)

DBUG Control Commands

a. Trace information can be inspected when the TRACE command has been specified through the predefined variable %TRACE. (3.3.1.2.e)

b. The FLOW option for the TRACE command is not available without the compiler parameter DEBUG (BREAK). (3.2.4.1)

c. At a breakpoint in the middle of a declarative section, the user can only refer to those declarations that have already been elaborated. (3.3.1.2.c)

Information Commands

a. The BASE option of the PUT and PUT_SCOPE commmands may be an integer from two through sixteen.

b. When no base is specified, the variables (and the components of variables) are printed out in their own modes (integer, string, enumeral, etc.).

c. When a base is specified, all the components of a composite variable are printed in the specified base. (3.3.1.2.d)

(5) Predefined Variables

The following requirements, like those for DBUG commands, address only issues beyond the basic functionality of the variables.

a. When the user sets the %HI_PRIORITY DBUG variable, all tasks below or equal to the specified value will be suspended when control returns to the DCP. (3.2.4.2.1.d)

b. The %TRACE.NEST variable always contains information while the other components of %TRACE require a TRACE command to have been given with the proper keyword option or the option ALL. (3.3.1.2.e)

c. The %TRACE.FLOW variable is not available if the DEBUG => BREAK parameter was not specified at compile time. (3.3.1.2.e)

4.3 Acceptance Test Requirements

In this section the requirements to be verified by the Acceptance Tests are listed as well as a description of the testing sessions which will comprise the Acceptance Tests. These testing sessions will also be a part of the AIE system testing which will be performed by test personnel prior to Acceptance Tests.

The following requirements are specified in Section 3.7.6 of the AIE system specification (AIE(1)). These specify functional characteristics required of the debugger.

1. DBUG shall be capable of controlling the execution, examination, and modification of an Ada program.

2. DBUG shall process a control language that permits a user to specify a variety of actions to be taken at execution control points (breakpoints).

3. Breakpoints shall be based on Ada statements or labels, task activations, subprogram calls, returns, and exceptions.

4. When a program in execution is suspended at a breakpoints, a user shall be able to display, dump, or modify selected data values in either machine or scalar type representation, trace the flow of program control, as well as display the formal names and values of subprograms.

5. DBUG may be used either interactively or in batch mode via command scripts.

6. The design shall provide interfaces for functional simulation.

7. The design shall be compatible with a future APSE tool that permits control and debugging of embedded computer software executing on a target machine.

The acceptance tests for these requirements will consist of three debugger sessions covering requirements 1-5 and a paragraph describing conformance to requirements 6 and 7. To satisfy requirement 5, one of the sessions will be interactive and the other two will use command scripts. The interactive session will be a small test while the background debugger sessions will be more complex they will demonstrate the conformance to the first four requirements.

The interactive session will be done by testing personnel using a pre-constructed sequence of debugger commands expected results. The debugger will be invoked to examine the execution of a simple fucntion in which a single breakpoint will be set using the BEFORE command. At the suspension due to the breakpoint, a GOTO command will be used to cause execution to resume at another part of the program. The return value from the function will depend on the correct transfer of control. A correct return value will be interpreted to satisfy the requirement specifying interactive debugging. The correct processing of the GOTO command will also demonstrate the first part of requirement 1 specifying control of program execution.

The second debugger session will use prepared input scripts and will direct the output to a file to provide automatic testing of requirements 2 and 3 and to verify the second part of requirement 5. The program to be debugged will be designed to provide structures to facilitate the setting of the desired breakpoints and the raising of the desired exceptions. Various breakpoints commands will be given to satisfy each part of requirement 3. These individual requirements will be met by the following commands: BEFORE, LABEL, ON_ENTRY, ON_EXIT, EXCEPTIONS and ALL_EXCEPTIONS. At each breakpoint, a LIST command will be issued to print the targeted statement in the output file to verify the break, and then a PROCEED will be issued to resume program execution. The LIST and PROCEED commands will be considered to demonstrate a variety of actions which can be specified at a breakpoint as in requirement 2. After these commands are successfully processed and the correct statements are found in the output file, requirements 2 and 3 will have been verified.

The third debugging session will be designed to verify requirements 1 and 4. This session will also use an input script and an output file so that it can be done automatically. The debugged programed used to demonstrate the conformance will be sufficiently complex to provide the information and control flow necessary to verify the requirements. Breakpoints will be set to provide enough information so that the output from the commands can be verified by test personnel at a later time. The commands that will be used to demonstrate the first part of requirement 4, that of displaying, modifying, and dumping of data values, will be the assignment command, PUT, and PUT_SCOPE with the BASE parameter being used to change the printed representation. The rest of requirement 4 will be verified using various options of the TRACE command. These as a group will be considered to be the acceptance test for the last two parts of requirement 1.

The last two requirements above, 6 and 7, specify certain design issues to be met by the structure of the debugger. Requirement 6 specifying interfaces for functional simulation was met by the hooks which may be inserted into the object code of a program by the compiler. These hooks provide branches to the DBUG Support Routine where additional operations for functional simulation can be added. Requirement 7 specifying compatibility with a future tool was met by the modular design of the debugger. The DBUG Support Routine interfaces with the DBUG COMMAND PROCESSOR only through the UTILITIES PROCEDURES CPC. Small changes to the UTILITIES PROCEDURES and the movement of the DBUG Support Routines to a target machine allows debugging on the target.

There are also speed and size requirements specified in AIE(1), Section 3.2.1.1.

1. The debug support task linked into programs for debugging shall not exceed 256K bytes.

2. The average time to start printing a simple Ada scalar variable after a complete request by name is entered to the debugger while at a breakpoint shall not exceed 1/2 second.

The first requirement will be passed by an examination of the statistics produced by the compilation of the DBUG Support Routines since they are a separable module. The second requirement will be demonstrated by the performance of the debugging in a background debugging session. The input will contain a command to print the time, ten DBUG commands requesting the printing of a scalar variable from the program, and another command to printing of a scalar variable from the program, and another command to print the (real) time. Since the printing is all being redirected to a file, each request is processed immediately after the last printing request has been procesed. The debugger will have passed the requirement if the time taken was not greater than five (10/2) seconds.